HOT Compilation: Elaboration

*TA: Akiva Leffert - aleffert@andrew.cmu.edu

Out: Wednesday, October 25, 2005
Due: Wednesday, November 8, 2005 (before midnight)

1 Introduction

Elaboration is the first phase of a type-preserving compiler: it transforms external language (EL), which is just abstract syntax, into an internal language (IL), which is a genuine type theory with a static and dynamic semantics. This translation serves two purposes: (1) it gives meaning to EL programs, precisely explaining the semantics of features not amenable to the usual type-theoretic analysis, and (2) it provides us with a starting point from which to conduct further type-directed translations in our compiler.

In this assignment, you will implement an elaborator for a small subset of Standard ML. Your implementation will have to account for several "non-type-theoretic" features of ML, including type inference, identifier resolution in the presence of "opened" modules, and module signature ascription.

2 Overview

Your task is to create a file `convert/elaborate.sml` defining a structure `Elaborate` matching the signature `ELABORATE` found in `convert/elaborate-sig.sml`, reproduced here for your convenience.

```
signature ELABORATE =
  sig
    exception Elab of string
    exception Error of string

    type tcontext = (IL.label * IL.dec) list

    val elab_ty : tcontext -> EL.ty -> IL.con
    val elab_exp : tcontext -> EL.exp -> IL.exp * IL.con

    val elab_bindings : tcontext -> EL.binding list
                     -> IL.sbnd list * IL.sdec list
    val elab_specs : tcontext -> EL.spec list -> IL.sdec list

    val elab_strexp : tcontext -> EL.strexp -> IL.module * IL.signat
    val elab_sigexp : tcontext -> EL.sigexp -> IL.signat

    val elab_program : tcontext -> EL.binding list
                    -> IL.sbnd list * IL.sdec list
  end
```

*Originally prepared by William Lovas (Fall 2005)

You should implement these elaboration functions by following the inference rules in Section 5. Signal any elaboration errors by raising `Elab` with an appropriate error message; you may use the `Error` exception for internal consistency errors such as violated invariants or the occurrence of "impossible" conditions.

The files `el/el.sml` and `il/il.sml` define the `EL` and `IL` structures, which contain all of the datatypes your elaborator will manipulate. The `IL` structure also contains several utility functions for manipulating `IL` terms that you may find useful, including substitution; you will find its signature in `il-sig.sml`.

The file `il/ilstatic-sig.sml` describes the signature of the `ILStatic` structure, which contains a typechecker for our internal language as well as helper functions for type inference. Those functions will be explained in Section 4.

Finally `common/variable-sig.sml` describes the signature of the `Variable` structure, our abstract representation of variables; this structure contains functions for generating and comparing variables.

You may use the test harness interface in the `ElaborationTop` structure (see `elaborationtop-sig.sml`) to experiment with your implementation. Several interesting examples are included in the `examples.sml` file. A few simple examples to get you started are shown in Section 6.

Submit your code via AFS by copying `elaborate.sml` to the directory

```
/afs/andrew/course/15/501-819/submit/<your andrew id>/elaboration
```

Note: Your submission will be graded automatically; if you submit an `elaborate.sml` that fails to compile under SML/NJ 110.0.59 using this assignment's base distribution, we won't be able to grade it.

3 Details

Our SML syntax appears in Figures 1 and 2. This EL is a very stripped down version of the EL from class.

The *ty* category has a production *tyvar*, representing "ticked" type variables like `'a` and `'b`. Full Standard ML has some odd conventions regarding *tyvar*s that permit a kind of implicit polymorphism in the EL, but we won't be supporting that in this assignment. In particular, *tyvar*s will only be used for binding the arguments of a `type` declaration, like `type 'a t = 'a list`; you won't be expected to handle *tyvar*s appearing anywhere else, such as in typing constraints or value specifications.

The syntax for the IL you'll be elaborating to appears in Figures 3, 4, 5, and 6. This language is very similar to what we've seen in class, but a few things bear mentioning.

First, we have product kinds $\langle knd_1 \times \cdots \times knd_n \rangle$, inhabited by tuples of constructors $\langle con_1, \ldots, con_n \rangle$. These are used to pass arguments to type operators and polymorphic values all at once, rather than one at a time. They get used by the type inference helper functions.

Second, there are several base types including booleans, integers, and lists as well as operations over these base types. Although we never discussed these in class, they should be unsurprising. There are several functions in `common/literal.sml` that will provide you with the argument and result types of unary and binary operators. In the rules, we invoke two functions $O_1 : unop \to con \times con$ and $O_2 : binop \to con \times con \times con$. The first function is for unary operators and the second is for binary operators. The final element in the result tuple for each function is the result type for the operator.

The IL also includes an anonymous recursive function expression, written $\textbf{fix}\ f\ (x{:}con_1) : con_2 \Rightarrow exp$. This represents a potentially recursive function of type $con_1 \to con_2$ with a body *exp*; both f and x are bound in *exp*: x is the argument to the function, and f represents the function itself. These are used for elaborating any kind of function, whether it can be recursive or not; when the function is not recursive, we write "_" for its fixed-point variable.

The IL syntax for constructors in Figure 3 does not include existential variables (evars), which are a special kind of type variable that occurs during inference. They do appear in the datatype `IL.con`. Your elaborator will concurrently perform type inference. However, as mentioned, we provide several functions that hide many of the gory details of type inference from you. Those are explained in Section 4 along with notes at relevant inference rules.

A full declarative static semantics for the IL appears in Appendix A.

The elaboration rules you'll be implementing are given in Section 5. They're grouped by judgment, and each judgment will correspond roughly to a function in your elaborator.

The static semantics uses typing contexts Γ while elaboration uses elaboration contexts Φ, the latter being endowed with labels. Their syntax is shown in Figure 7.

Elaboration makes use of a few derived forms, shown in Figure 8.

And now, without further ado, Elaboration!

longid	::= *id*	bare identifiers
	| *id* . *longid*	qualified identifiers
ty	::= `bool`	booleans
	| `unit`	unit
	| `int`	integers
	| `string`	strings
	| `char`	characters
	| ty_1 `*` ty_2	pairs
	| *ty* `list`	lists
	| ty_1 `->` ty_2	functions
	| `(`ty_1`,` ... `,`ty_n`)` *longid*	defined type constructors
	| *tyvar*	type variables
unop	::= `print`	string output
	| `int_to_string`	type conversion
	| `char_to_int`	type conversion
	| `char_to_string`	type conversion
binop	::= `+`	addition
	| `-`	subtraction
	| `*`	multiplication
	| `=`	integer equality
	| ∧	string concatenation
expr	::= *longid*	variables
	| `true`	boolean literals
	| `false`	
	| `if` $expr_1$ `then` $expr_2$ `else` $expr_3$	boolean test
	| ()	integer literals
	| $\bar{n}$	unit literals
	| *"string"*	string literals
	| *'c'*	character literals
	| *unop expr*	unary operations
	| $expr_1$ *binop* $expr_2$	binary operations
	| `(`$expr_1$`,` $expr_2$`)`	pairing
	| `#1` *expr*	first projection
	| `#2` *expr*	second projection
	| `nil`	empty list
	| $expr_1$ `::` $expr_2$	non-empty list
	| `case` *expr* `of nil =>` $expr_n$ `|` id_x `::` id_{xs} `=>` $expr_c$	list destructor
	| `fn` *id* `=>` *expr*	abstraction
	| $expr_1$ $expr_2$	application
	| `let` *bindings* `in` *expr* `end`	let-binding
	| *expr* `:` *ty*	constrained expressions

Figure 1: EL syntax, core language (except bindings)

binding ::=	`val` *id* `=` *e*	value binding
	| `fun` id_f id_x `=` *expr*	recursive function binding
	| `type (`$tyvar_1$`, ... ,`$tyvar_n$`)` *id* `=` *ty*	type binding
	| `open` *longid*	structure opening
	| `local` $bindings_1$ `in` $bindings_2$ `end`	local bindings
	| `structure` *id* `=` *strexp*	structure binding
bindings ::=	$\cdot$ | *binding bindings*	
strexp ::=	*longid*	structure identifiers
	| `struct` *bindings* `end`	structures
	| *strexp* `:>` *sigexp*	opaque signature ascription
	| *strexp* `:` *sigexp*	transparent signature ascription
	| `let` *bindings* `in` *strexp* `end`	module-level let-binding
sigexp ::=	`sig` *specs* `end`	structure signature
spec ::=	`val` *id* `:` *ty*	value specification
	| `type (`$tyvar_1$`, ... ,`$tyvar_n$`)` *id* `=` *ty*	transparent type specification
	| `type (`$tyvar_1$`, ... ,`$tyvar_n$`)` *id*	abstract type specification
	| `structure` *id* `:` *sigexp*	structure specification
specs ::=	$\cdot$ | *spec specs*	

Figure 2: EL syntax, modules and signatures

knd	$::= \Omega$	kind of types
	$\mid \Pi\alpha{:}knd_1.\, knd_2$	dependent function kinds
	$\mid \langle knd_1 \times \ldots \times knd_n \rangle$	product kinds
	$\mid \mathcal{S}(con)$	singleton kinds
con	$::= \alpha$	constructor variables
	$\mid \lambda\alpha{:}knd.\, con$	constructor functions
	$\mid con_1\ con_2$	constructor application
	$\mid \langle con_1, \ldots, con_n \rangle$	constructor tuples
	$\mid \pi_i\ con$	constructor tuple projection
	$\mid mod_{\mathrm{v}}.lab$	module projection
	$\mid \mathsf{bool}$	booleans
	$\mid \mathsf{unit}$	unit
	$\mid \mathsf{int}$	integers
	$\mid \mathsf{string}$	strings
	$\mid \mathsf{char}$	characters
	$\mid con_1 \times con_2$	pairs
	$\mid \mathsf{list}\ con$	lists
	$\mid con_1 \rightarrow con_2$	functions
	$\mid \forall\alpha{:}knd.\, con$	polymorphic types

Figure 3: IL syntax, kinds and constructors

exp	$::= x$	variables
	$\mid$ true	boolean literals
	$\mid$ false	
	$\mid$ **if** exp_1 **then** exp_2 **else** exp_3	boolean test
	$\mid ()$	unit literals
	$\mid \bar{n}$	integer literals
	$\mid$ *"string"*	string literals
	$\mid$ *'c'*	character literals
	$\mid unop\ exp$	unary operations
	$\mid exp_1\ binop\ exp_2$	binary operations
	$\mid (exp_1, exp_2)$	pairing
	$\mid \pi_1\ exp$	first projection
	$\mid \pi_2\ exp$	second projection
	$\mid$ nil $[con]$	empty list
	$\mid exp_1 :: exp_2$	non-empty list
	$\mid$ **case** exp **of** nil $\Rightarrow exp_n \mid x :: xs \Rightarrow exp_c$	list destructor
	$\mid$ **fix** $f\ (x{:}con_1) : con_2 \Rightarrow exp$	anonymous recursive function
	$\mid exp_1\ exp_2$	application
	$\mid \Lambda\alpha{:}knd.\ exp$	polymorphic abstraction
	$\mid exp\,[con]$	polymorphic application
	$\mid$ **let** $s = mod$ **in** exp **end**	module let-binding
	$\mid mod_{\mathrm{v}}.lab$	module projection

Figure 4: IL syntax, expressions

mod	$::= s$	module variables
	$\mid [sbnds]$	structures
	$\mid mod_{\mathrm{v}}.lab$	module projection
	$\mid mod :> sig$	sealed modules
$sbnds$	$::= \cdot \mid sbnd, sbnds$	
$sbnd$	$::= lab \triangleright bnd$	structure field binding
bnd	$::= x = exp$	expression variable binding
	$\mid \alpha = con$	constructor variable binding
	$\mid s = mod$	module variable binding
sig	$::= [sdecs]$	structure signature
$sdecs$	$::= \cdot \mid sdec, sdecs$	
$sdec$	$::= lab \triangleright dec$	structure field declaration
dec	$::= x{:}con$	expression variable declaration
	$\mid \alpha{:}knd$	constructor variable declaration
	$\mid s{:}sig$	module variable declaration

Figure 5: IL syntax, modules and signatures

$$
\begin{array}{rll}
exp_{\mathrm{v}} ::= & exppath & \text{expression paths} \\
| & \mathsf{true} & \text{boolean literals} \\
| & \mathsf{false} & \\
| & () & | \ \bar{n}\text{integer literals} \\
| & (exp_{\mathrm{v}1}, exp_{\mathrm{v}2}) & \text{pairs of values} \\
| & \mathsf{nil}\,[con] & \text{empty list} \\
| & exp_{\mathrm{v}1} :: exp_{\mathrm{v}2} & \text{lists of values} \\
| & \mathbf{fix}\ f\ (x{:}con_1) : con_2 \Rightarrow exp & \text{anonymous functions} \\
| & \Lambda\alpha{:}knd.\ exp_{\mathrm{v}} & \text{polymorphic values} \\
\\
mod_{\mathrm{v}} ::= & s & \text{module variables} \\
| & [sbnds_{\mathrm{v}}] & \text{structures of values} \\
| & mod_{\mathrm{v}}.lab & \text{projection from module values} \\
\\
sbnds_{\mathrm{v}} ::= & \cdot & \\
| & sbnds_{\mathrm{v}}, sbnd_{\mathrm{v}} & \\
\\
sbnd_{\mathrm{v}} ::= & lab \triangleright bnd_{\mathrm{v}} & \\
\\
bnd_{\mathrm{v}} ::= & x = exp_{\mathrm{v}} & \\
| & \alpha = con & \\
| & s = mod_{\mathrm{v}} &
\end{array}
$$

$$
\begin{array}{rll}
exppath ::= & x \ | \ modpath.lab & \text{paths} \\
conpath ::= & \alpha \ | \ modpath.lab & \\
modpath ::= & s \ | \ modpath.lab &
\end{array}
$$

$$
labs ::= lab \ | \ labs.lab \qquad \text{label sequences for identifier lookup}
$$

Figure 6: IL values and paths

$$
\begin{array}{rl}
\Gamma ::= \cdot & \text{empty typing context} \\
\mid \Gamma, x{:}con & \text{expression variable declaration} \\
\mid \Gamma, \alpha{:}knd & \text{constructor variable declaration} \\
\mid \Gamma, s{:}mod & \text{module variable declaration}
\end{array}
$$

$$
\begin{array}{rl}
\Phi ::= \cdot & \text{empty elaboration context} \\
\mid \Phi, lab \triangleright x{:}con & \text{expression variable declaration} \\
\mid \Phi, lab \triangleright \alpha{:}knd & \text{constructor variable declaration} \\
\mid \Phi, lab \triangleright s{:}sig & \text{module variable declaration}
\end{array}
$$

$$
\begin{array}{rl}
\cdot^\dagger \stackrel{\text{def}}{=} \cdot & \text{label erasure } (\Phi \rightsquigarrow \Gamma) \\
(\Phi, lab \triangleright dec)^\dagger \stackrel{\text{def}}{=} \Phi^\dagger, dec &
\end{array}
$$

Figure 7: Syntax of contexts

$$
\begin{array}{rcll}
knd^n & \stackrel{\text{def}}{=} & \langle knd \times \cdots (n \text{ times}) \cdots \times knd \rangle & \\
knd_1 \rightarrow knd_2 & \stackrel{\text{def}}{=} & \Pi\_{:}knd_1.\, knd_2 & \\
\lambda(\alpha_1, \ldots, \alpha_n).\, con & \stackrel{\text{def}}{=} & \lambda\alpha{:}\Omega^n.\, [\pi_1\, \alpha/\alpha_1] \cdots [\pi_n\, \alpha/\alpha_n] con & (\alpha \notin FV(con)) \\
\Pi(\alpha_1, \ldots, \alpha_n).\, knd & \stackrel{\text{def}}{=} & \Pi\alpha{:}\Omega^n.\, [\pi_1\, \alpha/\alpha_1] \cdots [\pi_n\, \alpha/\alpha_n] knd & (\alpha \notin FV(knd))
\end{array}
$$

Figure 8: Elaborator derived forms

4 Inference

Type inference won't be covered in class. However, a proper implementation of elaboration does inference as it goes. Therefore we provide you with a small library you'll need to use in your elaborator to handle inference. There are four functions and they are found in the `Inference` substructure of the `ILStatic` module. These functions deal with the process of making up types and turning them into concrete types, that is, inference. This substructure has the following signature which is found in `il/inference-sig.sml`:

```
signature INFERENCE =
  sig
    exception Equiv
    exception Occurs

    type context = IL.dec list
    type tcontext = (IL.label * IL.dec) list

    val new_con : tcontext -> IL.con
    val instantiate : tcontext -> (IL.exp * IL.con) -> (IL.exp * IL.con)
    val generalize : tcontext -> (tcontext -> IL.exp * IL.con)
                              -> IL.exp * IL.con
    val unify : context -> IL.con -> IL.con -> IL.kind -> unit
  end
```

Explanations for when these functions should be used and a rough outline of their purpose follow. Don't be too worried about what they do or how they work, but make sure you understand when they need to be called.

The `new_con` function makes up a new type. You will need to do this in any case where the type cannot determined deterministically. For example, you would elaborate `EL.ExNil` to `IL.ExNil(new_con ctx)` where `ctx` is the current elaboration context. Here you know it has some type, but you have no way of knowing what that type will be until later. Types created with this function will sometimes show up as something like `E1[{a, b}{s}]`[1] if you call one of the `ILPrint` functions from your code. However, if you see such a type in printed output after elaboration is done then you are improperly using the `generalize` function which will be described later in this section.

In some places you will need to assert that two constructors are equal. For example, when elaborating an `EL.ExIf` expression, you want to say that both arms have the same type. You can do this with the `unify` function. This function will attempt to unify the constructors and will raise an `Equiv` or `Occurs` exception if it fails. Make sure your code catches those exceptions and raises an `Elab` exception if necessary. This corresponds to usage of the $\Gamma \vdash con_1 \equiv con_2 : knd$ judgment.

The `instantiate` function should be called whenever you elaborate an expression variable. This function takes an expression and its type and, for polymorphic values, instantiates its type variables. It should be passed the expression and constructor returned for that variable by identifier lookup. In cases where identifier lookup fails you should raise an exception and thus not call this function.

Finally, the `generalize` function implements a wrapper function that handles polymorphic generalization. You should call the expression at an expression binding site, that is, when elaborating `BndVal` and `BndFun`. To use this function, pass it a function for elaborating the expression being bound and the translation context under which the expression should be elaborated. It will call your function and return an expression and constructor which you should use as the actual result of that elaboration function. Note: This function ignores the value restriction so don't be surprised by unusual behavior around effectful code.

[1] The sets in braces are the free constructor variables and the free module variables for the scope where the unification variable is instantiated. They are necessary for marking scope during generalization.

5 Elaboration

5.1 Types

$$\boxed{\Phi \vdash ty \rightsquigarrow con : \Omega}$$

$$\frac{}{\Phi \vdash \texttt{bool} \rightsquigarrow \mathsf{bool} : \Omega} \tag{5.1}$$

$$\frac{}{\Phi \vdash \texttt{unit} \rightsquigarrow \mathsf{unit} : \Omega} \tag{5.2}$$

$$\frac{}{\Phi \vdash \texttt{int} \rightsquigarrow \mathsf{int} : \Omega} \tag{5.3}$$

$$\frac{}{\Phi \vdash \texttt{string} \rightsquigarrow \mathsf{string} : \Omega} \tag{5.4}$$

$$\frac{}{\Phi \vdash \texttt{char} \rightsquigarrow \mathsf{char} : \Omega} \tag{5.5}$$

$$\frac{\Phi \vdash ty_1 \rightsquigarrow con_1 : \Omega \qquad \Phi \vdash ty_2 \rightsquigarrow con_2 : \Omega}{\Phi \vdash ty_1 \ \texttt{*} \ ty_2 \rightsquigarrow con_1 \times con_2 : \Omega} \tag{5.6}$$

$$\frac{\Phi \vdash ty \rightsquigarrow con : \Omega}{\Phi \vdash ty \ \texttt{list} \rightsquigarrow \mathsf{list}\, con : \Omega} \tag{5.7}$$

$$\frac{\Phi \vdash ty_1 \rightsquigarrow con_1 : \Omega \qquad \Phi \vdash ty_2 \rightsquigarrow con_2 : \Omega}{\Phi \vdash ty_1 \ \texttt{->} \ ty_2 \rightsquigarrow con_1 \to con_2 : \Omega} \tag{5.8}$$

$$\frac{\Phi \vdash ty_i \rightsquigarrow con_i : \Omega^n_{i=1} \qquad \Phi \vdash_{\overline{\mathrm{ctx}}} \overline{longid} \rightsquigarrow conpath : \Omega^n \to \Omega}{\Phi \vdash \texttt{(}ty_1\texttt{,} \ldots \texttt{,}ty_n\texttt{)} \ longid \rightsquigarrow conpath \ \langle con_1, \ldots, con_n \rangle : \Omega} \tag{5.9}$$

$$\frac{\Phi \vdash_{\overline{\mathrm{ctx}}} \overline{tyvar} \rightsquigarrow conpath : \Omega}{\Phi \vdash tyvar \rightsquigarrow conpath : \Omega} \tag{5.10}$$

5.2 Expressions

$$\boxed{\Phi \vdash expr \rightsquigarrow exp : con}$$

$$\frac{\Phi \vdash_{\overline{\mathrm{ctx}}} \overline{longid} \rightsquigarrow exppath : con}{\Phi \vdash longid \rightsquigarrow exppath : con} \tag{5.11}$$

Rule (5.11): You'll want to call `instantiate` on the result of identifier lookup.

$$\frac{}{\Phi \vdash \texttt{true} \rightsquigarrow \mathsf{true} : \mathsf{bool}} \tag{5.12}$$

$$\frac{}{\Phi \vdash \texttt{false} \rightsquigarrow \mathsf{false} : \mathsf{bool}} \tag{5.13}$$

$$\frac{\begin{array}{c}\Phi \vdash expr_1 \rightsquigarrow exp_1 : con_1 \qquad \Phi^\dagger \vdash con_1 \equiv \mathsf{bool} : \Omega \\ \Phi \vdash expr_2 \rightsquigarrow exp_2 : con_2 \qquad \Phi \vdash expr_3 \rightsquigarrow exp_3 : con_3 \\ \Phi^\dagger \vdash con_2 \equiv con_3 : \Omega\end{array}}{\Phi \vdash \texttt{if} \ expr_1 \ \texttt{then} \ expr_2 \ \texttt{else} \ expr_3 \rightsquigarrow \textbf{if} \ exp_1 \ \textbf{then} \ exp_2 \ \textbf{else} \ exp_3 : con_2} \tag{5.14}$$

$$\frac{}{\Phi \vdash () \rightsquigarrow () : \mathsf{unit}} \tag{5.15}$$

$$\frac{}{\Phi \vdash \bar{n} \rightsquigarrow \bar{n} : \mathsf{int}} \tag{5.16}$$

$$\frac{}{\Phi \vdash \text{“}string\text{”} \rightsquigarrow \text{“}string\text{”} : \mathsf{string}} \tag{5.17}$$

$$\frac{}{\Phi \vdash \text{‘}c\text{’} \rightsquigarrow \text{‘}c\text{’} : \mathsf{char}} \tag{5.18}$$

$$\frac{O_1(unop) = (con, con_r) \qquad \Phi \vdash expr \rightsquigarrow exp : con' \qquad \Phi^\dagger \vdash con \equiv con' : \Omega}{\Phi \vdash unop\ expr \rightsquigarrow unop\ exp : con_r} \tag{5.19}$$

$$\frac{O_2(binop) = (con_1, con_2, con_r) \qquad \Phi \vdash expr_1 \rightsquigarrow exp_1 : con_1' \qquad \Phi \vdash expr_2 \rightsquigarrow exp_2 : con_2' \qquad \Phi^\dagger \vdash con_1 \equiv con_1' : \Omega \qquad \Phi^\dagger \vdash con_2 \equiv con_2' : \Omega}{\Phi \vdash expr_1\ binop\ expr_2 \rightsquigarrow exp_1\ binop\ exp_2 : con_r} \tag{5.20}$$

$$\frac{\Phi \vdash expr_1 \rightsquigarrow exp_1 : con_1 \qquad \Phi \vdash expr_2 \rightsquigarrow exp_2 : con_2}{\Phi \vdash (expr_1\texttt{,}\ expr_2) \rightsquigarrow (exp_1, exp_2) : con_1 \times con_2} \tag{5.21}$$

$$\frac{\Phi \vdash expr \rightsquigarrow exp : con \qquad \Phi^\dagger \vdash con \equiv con_1 \times con_2 : \Omega}{\Phi \vdash \texttt{\#1}\ expr \rightsquigarrow \pi_1\ exp : con_1} \tag{5.22}$$

$$\frac{\Phi \vdash expr \rightsquigarrow exp : con \qquad \Phi^\dagger \vdash con \equiv con_1 \times con_2 : \Omega}{\Phi \vdash \texttt{\#2}\ expr \rightsquigarrow \pi_2\ exp : con_2} \tag{5.23}$$

$$\frac{\Phi^\dagger \vdash con : \Omega}{\Phi \vdash \texttt{nil} \rightsquigarrow \mathsf{nil}\,[con] : \mathsf{list}\ con} \tag{5.24}$$

Rule (5.24): Your elaborator should create a new *con* for use with type inference.

$$\frac{\Phi \vdash expr_1 \rightsquigarrow exp_1 : con_1 \qquad \Phi \vdash expr_2 \rightsquigarrow exp_2 : con_2 \qquad \Phi^\dagger \vdash \mathsf{list}\ con_1 \equiv con_2 : \Omega}{\Phi \vdash expr_1\ \texttt{::}\ expr_2 \rightsquigarrow exp_1 :: exp_2 : \mathsf{list}\ con_1} \tag{5.25}$$

$$\frac{\begin{array}{c}\Phi \vdash expr \rightsquigarrow exp : con \qquad \Phi^\dagger \vdash con \equiv \mathsf{list}\ con' : \Omega \\ \Phi \vdash expr_n \rightsquigarrow exp_n : con_n \\ (x, xs \notin \mathrm{dom}(\Phi)) \qquad \Phi, \overline{id_x} \triangleright x{:}con', \overline{id_{xs}} \triangleright xs{:}\mathsf{list}\ con' \vdash expr_c \rightsquigarrow exp_c : con_c \qquad \Phi^\dagger \vdash con_n \equiv con_c : \Omega\end{array}}{\begin{array}{c}\Phi \vdash \texttt{case}\ expr\ \texttt{of nil =>}\ expr_n\ \texttt{|}\ id_x\ \texttt{::}\ id_{xs}\ \texttt{=>}\ expr_c \rightsquigarrow \\ \textbf{case}\ exp\ \textbf{of}\ \mathsf{nil} \Rightarrow exp_n \mid x :: xs \Rightarrow exp_c : con_n\end{array}} \tag{5.26}$$

$$\frac{\Phi^\dagger \vdash con : \Omega \qquad \Phi, \overline{id} \triangleright x{:}con \vdash expr \rightsquigarrow exp : con' \qquad (x \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{fn}\ id\ \texttt{=>}\ expr \rightsquigarrow \textbf{fix}\ \_\ (x{:}con) : con' \Rightarrow exp : con \to con'} \tag{5.27}$$

Rule (5.27): The IL only provides a primitive for recursive anonymous functions; the use of “_” here as the recursive bound variable is simply meant to suggest that the function is not in fact recursive. Your elaborator should create a new *con* for use with type inference.

$$\frac{\Phi \vdash expr_1 \rightsquigarrow exp_1 : con_1 \qquad \Phi \vdash expr_2 \rightsquigarrow exp_2 : con_2 \qquad \Phi^\dagger \vdash con_1 \equiv con_2 \to con : \Omega}{\Phi \vdash expr_1\ expr_2 \rightsquigarrow exp_1\ exp_2 : con} \tag{5.28}$$

$$\frac{\Phi \vdash bindings \rightsquigarrow sbnds : sdecs \qquad (s \notin \mathrm{dom}(\Phi)) \qquad \Phi, 1^\star \triangleright s{:}[sdecs] \vdash expr \rightsquigarrow exp : con' \qquad (\Phi, 1^\star \triangleright s{:}[sdecs])^\dagger \vdash con' \equiv con : \Omega \qquad (s \notin FV(con))}{\Phi \vdash \texttt{let}\ bindings\ \texttt{in}\ expr\ \texttt{end} \rightsquigarrow \textbf{let}\ s = [sbnds]\ \textbf{in}\ exp\ \textbf{end} : con} \tag{5.29}$$

Rule (5.29): We find a type con that is equivalent to con' but doesn't depend on any abstract types defined in the let-bindings. This can be done, for example, by normalizing con', but for this assignment you can simply use con', and check that it does not depend on s.

$$\frac{\Phi \vdash expr \rightsquigarrow exp : con' \qquad \Phi \vdash ty \rightsquigarrow con : \Omega \qquad \Phi^\dagger \vdash con \equiv con' : \Omega}{\Phi \vdash (expr\ :\ ty) \rightsquigarrow exp : con} \tag{5.30}$$

5.3 Bindings

Elaboration of bindings makes use of a "syntactic concatenation with renaming" operation, defined in parallel on *sbnds* and *sdecs* as follows:

$$(\cdot +\!+ sbnds) : (\cdot +\!+ sdecs) \stackrel{\text{def}}{=} sbnds : sdecs$$

$$(lab \triangleright bnd, sbnds +\!+ sbnds') : (lab \triangleright dec, sdecs +\!+ sdecs') \stackrel{\text{def}}{=}$$

$$\begin{cases} lab \triangleright bnd, sbnds'' : lab \triangleright dec, sdecs'' & \text{if } lab \notin \mathrm{labdom}(sbnds'') \\ lab' \triangleright bnd, sbnds'' : lab' \triangleright dec, sdecs'' & \text{otherwise, where } lab' \notin \mathrm{labdom}(sbnds'') \\ & (lab' \text{ internal}) \end{cases}$$

where $sbnds +\!+ sbnds' : sdecs +\!+ sdecs' = sbnds'' : sdecs''$

In words, $sbnds +\!+ sbnds' : sdecs +\!+ sdecs'$ concatenates $sbnds$ with $sbnds'$ and $sdecs$ with $sdecs'$ while renaming labels in $sbnds : sdecs$ that are shadowed by labels in $sbnds' : sdecs'$. Such shadowed labels are given fresh labels in order to render them inaccessible to the programmer. The labdom function just calculates the set of labels of an *sbnds*.

$\boxed{\Phi \vdash bindings \rightsquigarrow sbnds : sdecs}$

$$\frac{}{\Phi \vdash \cdot \rightsquigarrow \cdot : \cdot} \tag{5.31}$$

$$\frac{\Phi \vdash binding \rightsquigarrow sbnds_1 : sdecs_1 \qquad \Phi, sdecs_1 \vdash bindings \rightsquigarrow sbnds_2 : sdecs_2}{\Phi \vdash binding\ bindings \rightsquigarrow sbnds_1 +\!+ sbnds_2 : sdecs_1 +\!+ sdecs_2} \tag{5.32}$$

$\boxed{\Phi \vdash binding \rightsquigarrow sbnds : sdecs}$

$$\frac{\Phi \vdash expr \rightsquigarrow exp : con \qquad (x \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{val}\ id\ \texttt{=}\ expr \rightsquigarrow \overline{id} \triangleright x{=}exp : \overline{id} \triangleright x{:}con} \tag{5.33}$$

Rule (5.33): Here you'll want to call the `generalize` function with your function to elaborate *exp* and *con*.

$$\frac{\Phi^\dagger \vdash con \to con' : \Omega \qquad (f, x \notin \mathrm{dom}(\Phi)) \qquad \Phi, \overline{id_f} \triangleright f{:}con \to con', \overline{id_x} \triangleright x{:}con \vdash expr \rightsquigarrow exp : con'' \qquad \Phi^\dagger \vdash con' \equiv con'' : \Omega \qquad (f' \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{fun}\ id_f\ id_x\ \texttt{=}\ expr \rightsquigarrow \overline{id_f} \triangleright f'{=}\textbf{fix}\ f\ (x{:}con) : con' \Rightarrow exp : \overline{id_f} \triangleright f'{:}con \to con'} \tag{5.34}$$

Rule (5.34): As with `val` binding above, you should wrap your implementation with the `generalize` function.

$$\frac{(\alpha_1, \ldots, \alpha_n \notin \mathrm{dom}(\Phi)) \qquad \Phi, \overline{tyvar_1} \triangleright \alpha_1{:}\Omega, \ldots, \overline{tyvar_n} \triangleright \alpha_n{:}\Omega \vdash ty \rightsquigarrow con : \Omega \qquad (\alpha \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{type (}tyvar_1\texttt{, } \ldots \texttt{, }tyvar_n\texttt{) } id \texttt{ = } ty \rightsquigarrow \overline{id} \triangleright \alpha{=}\lambda(\alpha_1, \ldots, \alpha_n).\, con : \overline{id} \triangleright \alpha{:}\Pi(\alpha_1, \ldots, \alpha_n).\, \mathcal{S}(con)} \tag{5.35}$$

$$\frac{\Phi \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow modpath : sig \qquad (s \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{open } longid \rightsquigarrow 1^\star \triangleright s{=}modpath : 1^\star \triangleright s{:}sig} \tag{5.36}$$

$$\frac{\Phi \vdash bindings_1 \rightsquigarrow sbnds_1 : sdecs_1 \qquad (s \notin \mathrm{dom}(\Phi)) \qquad \Phi, 1^\star \triangleright s{:}[sdecs_1] \vdash bindings_2 \rightsquigarrow sbnds_2 : sdecs_2}{\Phi \vdash \texttt{local } bindings_1 \texttt{ in } bindings_2 \texttt{ end} \rightsquigarrow 1 \triangleright s{=}[sbnds_1], sbnds_2 : 1 \triangleright s{:}[sdecs_1], sdecs_2} \tag{5.37}$$

Rule (5.37): We use the starred structure convention to allow access to $bindings_1$'s bindings while elaborating $bindings_2$. While elaborating $bindings_2$, identifier lookup will have resolved any references to bindings from $bindings_1$; thus we can remove the star from our output so that no other outside code can access those local bindings.

$$\frac{\Phi \vdash strexp \rightsquigarrow mod : sig \qquad (s \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{structure } id \texttt{ = } strexp \rightsquigarrow \overline{id} \triangleright s{=}mod : \overline{id} \triangleright s{:}sig} \tag{5.38}$$

5.4 Structure Expressions

$$\boxed{\Phi \vdash strexp \rightsquigarrow mod : sig}$$

$$\frac{\Phi \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow modpath : sig}{\Phi \vdash longid \rightsquigarrow modpath : sig} \tag{5.39}$$

$$\frac{\Phi \vdash bindings \rightsquigarrow sbnds : sdecs}{\Phi \vdash \texttt{struct } bindings \texttt{ end} \rightsquigarrow [sbnds] : [sdecs]} \tag{5.40}$$

$$\frac{\Phi \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow modpath : sig \qquad \Phi \vdash sigexp \rightsquigarrow sig' : \mathbf{Sig} \qquad \Phi \vdash_{\mathrm{sub}} modpath : sig \preceq sig' \rightsquigarrow mod : sig''}{\Phi \vdash longid \texttt{ :> } sigexp \rightsquigarrow (mod :> sig') : sig'} \tag{5.41}$$

$$\frac{\Phi \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow modpath : sig \qquad \Phi \vdash sigexp \rightsquigarrow sig' : \mathbf{Sig} \qquad \Phi \vdash_{\mathrm{sub}} modpath : sig \preceq sig' \rightsquigarrow mod : sig''}{\Phi \vdash longid \texttt{ : } sigexp \rightsquigarrow mod : sig''} \tag{5.42}$$

Rules (5.41) and (5.42): Opaque signature ascription may hide some type equalities through the use of IL module sealing. Transparent signature ascription exposes as much type equality information as possible.

$$\frac{(strexp \neq longid) \qquad (id \text{ fresh}) \qquad \Phi \vdash \texttt{let structure } id \texttt{ = } strexp \texttt{ in } id \texttt{ :> } sigexp \texttt{ end} \rightsquigarrow mod : sig}{\Phi \vdash strexp \texttt{ :> } sigexp \rightsquigarrow mod : sig} \tag{5.43}$$

$$\frac{(strexp \neq longid) \qquad (id \text{ fresh}) \qquad \Phi \vdash \texttt{let structure } id \texttt{ = } strexp \texttt{ in } id \texttt{ : } sigexp \texttt{ end} \rightsquigarrow mod : sig}{\Phi \vdash strexp \texttt{ : } sigexp \rightsquigarrow mod : sig} \tag{5.44}$$

Rules (5.43) and (5.44): Treat sealed non-identifiers as a derived form.

$$\frac{\begin{array}{c}\Phi \vdash \mathit{bindings} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs} \\ (s \notin \mathrm{dom}(\Phi)) \qquad \Phi, \mathrm{private}^\star \triangleright s{:}[\mathit{sdecs}] \vdash \mathit{strexp} \rightsquigarrow \mathit{mod} : \mathit{sig} \qquad (s' \notin \mathrm{dom}(\Phi))\end{array}}{\begin{array}{c}\Phi \vdash \texttt{let}\ \mathit{bindings}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \rightsquigarrow \\ [\mathrm{private} \triangleright s{=}[\mathit{sbnds}], \mathrm{public}^\star \triangleright s'{=}\mathit{mod}] : [\mathrm{private} \triangleright s{:}[\mathit{sdecs}], \mathrm{public}^\star \triangleright s'{:}\mathit{sig}]\end{array}} \tag{5.45}$$

Rule (5.45): Recall from class how we use a structure with an inaccessible component to sidestep the avoidance problem. While elaborating the *strexp*, identifier lookup will look inside s to find identifiers from *bindings*; in the output, we remove the $\star$ from the private component to render *bindings* inaccessible from the outside. See also Rule (5.37).

5.5 Signature Expressions

$\boxed{\Phi \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathbf{Sig}}$

$$\frac{\Phi \vdash \mathit{specs} \rightsquigarrow \mathit{sdecs}}{\Phi \vdash \texttt{sig}\ \mathit{specs}\ \texttt{end} \rightsquigarrow [\mathit{sdecs}] : \mathbf{Sig}} \tag{5.46}$$

5.6 Specifications

When elaborating a list of specifications, we must ensure that no duplicate identifiers are bound. To that end, we define a function vislabs(*sdecs*) that computes the set of labels visible to identifier lookup.

$$\begin{aligned}
\mathrm{vislabs}(\cdot) &\stackrel{\mathrm{def}}{=} \emptyset \\
\mathrm{vislabs}(\mathit{sdec}, \mathit{sdecs}) &\stackrel{\mathrm{def}}{=} \mathrm{vislabs}(\mathit{sdec}) \cup \mathrm{vislabs}(\mathit{sdecs}) \\
\\
\mathrm{vislabs}(\mathit{lab} \triangleright x{:}\mathit{con}) &\stackrel{\mathrm{def}}{=} \{\mathit{lab}\} \\
\mathrm{vislabs}(\mathit{lab} \triangleright \alpha{:}\mathit{knd}) &\stackrel{\mathrm{def}}{=} \{\mathit{lab}\} \\
\mathrm{vislabs}(\mathit{lab} \triangleright s{:}\mathit{sig}) &\stackrel{\mathrm{def}}{=} \{\mathit{lab}\} \qquad \text{if } \mathit{lab} \text{ not starred} \\
\mathrm{vislabs}(\mathit{lab}^\star \triangleright s{:}[\mathit{sdecs}]) &\stackrel{\mathrm{def}}{=} \mathrm{vislabs}(\mathit{sdecs})
\end{aligned}$$

$\boxed{\Phi \vdash \mathit{specs} \rightsquigarrow \mathit{sdecs}}$

$$\frac{}{\Phi \vdash \cdot \rightsquigarrow \cdot} \tag{5.47}$$

$$\frac{\begin{array}{c}\Phi \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}_1 \qquad \Phi, \mathit{sdecs}_1 \vdash \mathit{specs} \rightsquigarrow \mathit{sdecs}_2 \\ (\mathrm{vislabs}(\mathit{sdecs}_1) \cap \mathrm{vislabs}(\mathit{sdecs}_2) = \emptyset)\end{array}}{\Phi \vdash \mathit{spec}\ \mathit{specs} \rightsquigarrow \mathit{sdecs}_1, \mathit{sdecs}_2} \tag{5.48}$$

$\boxed{\Phi \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}}$

$$\frac{\Phi \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega \qquad (x \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{val}\ \mathit{id}\ \texttt{:}\ \mathit{ty} \rightsquigarrow \overline{\mathit{id}} \triangleright x{:}\mathit{con}} \tag{5.49}$$

$$\frac{(\alpha \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{type (}\mathit{tyvar}_1\texttt{, ... ,}\mathit{tyvar}_n\texttt{)}\ \mathit{id} \rightsquigarrow \overline{\mathit{id}} \triangleright \alpha{:}\Omega^n \to \Omega} \tag{5.50}$$

$$\frac{(\alpha_1, \ldots, \alpha_n \notin \mathrm{dom}(\Phi)) \qquad \Phi, \overline{\mathit{tyvar}}_1 \triangleright \alpha_1{:}\Omega, \ldots, \overline{\mathit{tyvar}}_n \triangleright \alpha_n{:}\Omega \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega \qquad (\alpha \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{type (}\mathit{tyvar}_1\texttt{, ... ,}\mathit{tyvar}_n\texttt{)}\ \mathit{id}\ \texttt{=}\ \mathit{ty} \rightsquigarrow \overline{\mathit{id}} \triangleright \alpha{:}\Pi(\alpha_1, \ldots, \alpha_n).\, \mathcal{S}(\mathit{con})} \tag{5.51}$$

$$\frac{\Phi \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathbf{Sig} \qquad (s \notin \mathrm{dom}(\Phi))}{\Phi \vdash \texttt{structure}\ \mathit{id}\ \texttt{:}\ \mathit{sigexp} \rightsquigarrow \overline{\mathit{id}} \triangleright s{:}\mathit{sig}} \tag{5.52}$$

5.7 Identifier Resolution

Identifier resolution makes use of a "starred structure convention" to account for open modules. The lookup judgment $\Phi \vdash_{\mathrm{ctx}} \mathit{labs} \rightsquigarrow \mathit{path} : \mathit{class}$ scans backwards through the context looking for the most recently bound label matching the long identifier, looking inside structures whose labels are starred, like $\mathit{lab}^\star$. Note that in order to simplify the presentation, we make use of the following neutral metavariables:

$$\begin{aligned} \mathit{path} &\text{ for one of } \mathit{exppath}, \mathit{conpath}, \mathit{modpath} \\ \mathit{class} &\text{ for one of } \mathit{con}, \mathit{knd}, \mathit{sig} \\ \mathit{var} &\text{ for one of } x, \alpha, s \end{aligned}$$

$$\boxed{\Phi \vdash_{\mathrm{ctx}} \mathit{labs} \rightsquigarrow \mathit{path} : \mathit{class}}$$

$$\frac{\Phi \vdash_{\mathrm{ctx}} \mathit{lab} \rightsquigarrow \mathit{path} \qquad \Phi^\dagger \vdash \mathit{path} : \mathit{class}}{\Phi \vdash_{\mathrm{ctx}} \mathit{lab} \rightsquigarrow \mathit{path} : \mathit{class}} \tag{5.53}$$

$$\frac{\Phi \vdash_{\mathrm{ctx}} \mathit{labs} \rightsquigarrow \mathit{modpath} : [\mathit{sdecs}] \qquad \mathit{sdecs} \vdash_{\mathrm{sig}} \mathit{lab} \rightsquigarrow \mathit{labs}' \qquad \Phi^\dagger \vdash \mathit{modpath}.\mathit{labs}' : \mathit{class}}{\Phi \vdash_{\mathrm{ctx}} \mathit{labs}.\mathit{lab} \rightsquigarrow \mathit{modpath}.\mathit{labs}' : \mathit{class}} \tag{5.54}$$

$$\boxed{\Phi \vdash_{\mathrm{ctx}} \mathit{lab} \rightsquigarrow \mathit{path}}$$

An auxiliary classifier-free judgment does the real work of actually finding the path corresponding to a label.

$$\frac{}{\Phi, \mathit{lab} \triangleright \mathit{var}{:}\mathit{class} \vdash_{\mathrm{ctx}} \mathit{lab} \rightsquigarrow \mathit{var}} \tag{5.55}$$

$$\frac{(\mathit{lab} \neq \mathit{lab}', \mathit{lab} \text{ is not starred}) \qquad \Phi \vdash_{\mathrm{ctx}} \mathit{lab}' \rightsquigarrow \mathit{path}}{\Phi, \mathit{lab} \triangleright \mathit{var}{:}\mathit{class} \vdash_{\mathrm{ctx}} \mathit{lab}' \rightsquigarrow \mathit{path}} \tag{5.56}$$

$$\frac{\mathit{sdecs} \vdash_{\mathrm{sig}} \mathit{lab}' \rightsquigarrow \mathit{labs}}{\Phi, \mathit{lab}^\star \triangleright s{:}[\mathit{sdecs}] \vdash_{\mathrm{ctx}} \mathit{lab}' \rightsquigarrow s.\mathit{labs}} \tag{5.57}$$

$$\frac{\mathit{sdecs} \vdash_{\mathrm{sig}} \mathit{lab}' \not\rightsquigarrow \qquad \Phi \vdash_{\mathrm{ctx}} \mathit{lab}' \rightsquigarrow \mathit{path}}{\Phi, \mathit{lab}^\star \triangleright s{:}[\mathit{sdecs}] \vdash_{\mathrm{ctx}} \mathit{lab}' \rightsquigarrow \mathit{path}} \tag{5.58}$$

$$\boxed{\Phi; \mathit{modpath} : \mathit{sig} \vdash_{\mathrm{sig}} \mathit{lab} \rightsquigarrow \mathit{path} : \mathit{class}}$$

$$\frac{\mathit{sdecs} \vdash_{\mathrm{sig}} \mathit{lab} \rightsquigarrow \mathit{labs} \qquad \Phi^\dagger \vdash \mathit{modpath}.\mathit{labs} : \mathit{class}}{\Phi; \mathit{modpath} : [\mathit{sdecs}] \vdash_{\mathrm{sig}} \mathit{lab} \rightsquigarrow \mathit{modpath}.\mathit{labs} : \mathit{class}} \tag{5.59}$$

$$\boxed{\mathit{sdecs} \vdash_{\mathrm{sig}} \mathit{lab} \rightsquigarrow \mathit{labs}}$$

Auxiliary classifier-free signature lookup judgment.

$$\frac{}{\mathit{sdecs}, \mathit{lab} \triangleright \mathit{var}{:}\mathit{class} \vdash_{\mathrm{sig}} \mathit{lab} \rightsquigarrow \mathit{lab}} \tag{5.60}$$

$$\frac{(\mathit{lab} \neq \mathit{lab}', \mathit{lab} \text{ is not starred}) \qquad \mathit{sdecs} \vdash_{\mathrm{sig}} \mathit{lab}' \rightsquigarrow \mathit{labs}}{\mathit{sdecs}, \mathit{lab} \triangleright \mathit{var}{:}\mathit{class} \vdash_{\mathrm{sig}} \mathit{lab}' \rightsquigarrow \mathit{labs}} \tag{5.61}$$

$$\frac{sdecs' \vdash_{\text{sig}} lab' \rightsquigarrow labs}{sdecs, lab^\star \triangleright s{:}[sdecs'] \vdash_{\text{sig}} lab' \rightsquigarrow lab^\star.labs} \tag{5.62}$$

$$\frac{sdecs' \vdash_{\text{sig}} lab' \not\rightsquigarrow \qquad sdecs \vdash_{\text{sig}} lab' \rightsquigarrow labs}{sdecs, lab^\star \triangleright s{:}[sdecs'] \vdash_{\text{sig}} lab' \rightsquigarrow labs} \tag{5.63}$$

5.8 Coercive Signature Matching

The IL subsignature relation only respects pointwise subkinding and subsignature matching, without accounting for any reordering or dropping of structure fields. Instead of including these in the IL's semantics, they're handled by the coercive signature matching judgment $\Phi \vdash_{\text{sub}} modpath : sig_1 \preceq sig_2 \rightsquigarrow mod : sig$. Given a module path $modpath$ with signature sig_1, this judgement projects components out of $modpath$ to produce a module mod matching signature sig_2, with principal signature sig.

Note: in full Standard ML, this judgment handles partial polymorphic instantiation, but we're not treating explicit polymorphic variables in this assignment, so we simplify things a bit.

$$\boxed{\Phi \vdash_{\text{sub}} modpath : sig_1 \preceq sig_2 \rightsquigarrow mod : sig}$$

$$\frac{\begin{array}{c}\Phi; modpath{:}sig \vdash_{\text{sub}} sdec_1 \rightsquigarrow sbnd_1 : sdec_1' \\ (\Phi, sdec_1'); modpath{:}sig \vdash_{\text{sub}} sdec_2 \rightsquigarrow sbnd_2 : sdec_2' \\ \vdots \\ (\Phi, sdec_1', \ldots, sdec_{n-1}'); modpath{:}sig \vdash_{\text{sub}} sdec_n \rightsquigarrow sbnd_n : sdec_n'\end{array}}{\Phi \vdash_{\text{sub}} modpath : sig \preceq [sdec_1, \ldots, sdec_n] \rightsquigarrow [sbnd_1, \ldots, sbnd_n] : [sdec_1', \ldots, sdec_n']} \tag{5.64}$$

$$\boxed{\Phi; modpath{:}sig \vdash_{\text{sub}} sdec \rightsquigarrow sbnd : sdec}$$

$$\frac{\Phi; modpath : [sdecs] \vdash_{\text{sig}} lab \rightsquigarrow exppath : con' \qquad \Phi^\dagger \vdash con' \equiv con : \Omega}{\Phi; modpath{:}[sdecs] \vdash_{\text{sub}} lab \triangleright x{:}con \rightsquigarrow lab \triangleright x{=}exppath : lab \triangleright x{:}con} \tag{5.65}$$

$$\frac{\Phi; modpath : [sdecs] \vdash_{\text{sig}} lab \rightsquigarrow conpath : knd' \qquad \Phi^\dagger \vdash knd' \leq knd}{\Phi; modpath{:}[sdecs] \vdash_{\text{sub}} lab \triangleright \alpha{:}knd \rightsquigarrow lab \triangleright \alpha{=}conpath : lab \triangleright \alpha{:}knd'} \tag{5.66}$$

$$\frac{\Phi; modpath : [sdecs] \vdash_{\text{sig}} lab \rightsquigarrow modpath' : sig' \qquad \Phi \vdash_{\text{sub}} modpath' : sig' \preceq sig \rightsquigarrow mod : sig''}{\Phi; modpath{:}[sdecs] \vdash_{\text{sub}} lab \triangleright s{:}sig \rightsquigarrow lab \triangleright s{=}mod : lab \triangleright s{:}sig''} \tag{5.67}$$

6 Examples

The following interactions with the SML/NJ top-level will give you some simple examples of how your elaborator should work. Note that your elaborator need not produce this output exactly, but it should produce something very similar. (The notion of "equivalence" for this assignment is different from just α-equivalence, since some elaboration rules require you to make up fresh labels, and labels do not α-vary.) More sample inputs may be found in the file `el/elexamples.sml` distributed with this project's base; your elaborator should at least accept all of these.

6.1 Val bindings:

```
- Top.elab_string "val x = 5";
  |- val x = 5
 ~~> x |> x_1 = 5
   : x |> x_1 : int
val it = () : unit
```

```
- Top.elab_string "val l = nil";
  |- val l = nil
 ~~> l |> l_3 = /\ a_2:<Type>. nil [#1 a_2]
   : l |> l_3 : All a_2:<Type>. list (#1 a_2)
val it = () : unit
```

6.2 Functions:

```
- Top.elab_string "fun g x = x + 7";
  |- fun g x = x + 7
 ~~> g |> g_9 = fix g_10 (x_11 : int) : int => x_11 + 7
   : g |> g_9 : int -> int
val it = () : unit

- Top.elab_string "fun f x = x";
  |- fun f x = x
 ~~> f |> f_5 = /\ a_4:<Type>. fix f_6 (x_7 : #1 a_4) : #1 a_4 => x_7
   : f |> f_5 : All a_4:<Type>. #1 a_4 -> #1 a_4
val it = () : unit

- Top.elab_string "val compose = fn f => fn g => fn x => f (g x)";
  |- val compose = fn f => fn g => fn x => f (g x)
 ~~> compose |> compose_1155 =
         /\ a_1148:<Type * Type * Type>.
         fix __1149 (f_1150 : #1 a_1148 -> #2 a_1148)
           : (#3 a_1148 -> #1 a_1148) -> #3 a_1148 -> #2 a_1148 =>
             fix __1151 (g_1152 : #3 a_1148 -> #1 a_1148)
               : #3 a_1148 -> #2 a_1148 =>
                 fix __1153 (x_1154 : #3 a_1148)
                   : #2 a_1148 =>
                     f_1150 (g_1152 x_1154)
   : compose |> compose_1155 :
         All a_1148:<Type * Type * Type>.
         (#1 a_1148 -> #2 a_1148)
         -> (#3 a_1148 -> #1 a_1148) -> #3 a_1148 -> #2 a_1148
val it = () : unit
```

6.3 Type bindings

```
- Top.elab_string "type t = int";
  |- type t = int
 ~~> t |> t_61 = \ var_62:<>. int
   : t |> t_61 : Pi var_63:<>. S(int)
val it = () : unit

- Top.elab_string "type ('a, 'b) t = 'a list -> 'b";
  |- type ('a, 'b) t = 'a list -> 'b
 ~~> t |> t_66 = \ var_67:<Type * Type>. list (#1 var_67) -> #2 var_67
   : t |> t_66 : Pi var_68:<Type * Type>. S(list (#1 var_68) -> #2 var_68)
val it = () : unit
```

6.4 Transparent versus opaque signature ascription:

```
- Top.elab_string "structure S : sig type t end = \
=                 \  struct                      \
=                 \    type t = int              \
=                 \  end";
  |- structure S = struct type t = int end : sig type t end
 ~~> S |> S_21 =
         [private |> s_16 =
              [%module_0 |> %module_0_15 = [t |> t_12 = \ var_13:<>. int]],
          public* |> __20 = [t |> t_17 = s_16.%module_0.t]]
   : S |> S_21 :
         [private |> s_16 :
              [%module_0 |> %module_0_15 : [t |> t_12 : Pi var_14:<>. S(int)]],
          public* |> __20 :
              [t |> t_17 : Pi var_14:<>. S(s_16.%module_0.t var_14)]]
val it = () : unit

- Top.elab_string "structure S :> sig type t end = \
=                 \  struct                       \
=                 \    type t = int               \
=                 \  end";
  |- structure S = struct type t = int end :> sig type t end
 ~~> S |> S_31 =
         [private |> s_26 =
              [%module_1 |> %module_1_25 = [t |> t_22 = \ var_23:<>. int]],
          public* |> __30 =
              [t |> t_27 = s_26.%module_1.t]
              :> [t |> t_27 : Pi __28:<>. Type]]
   : S |> S_31 :
         [private |> s_26 :
              [%module_1 |> %module_1_25 : [t |> t_22 : Pi var_24:<>. S(int)]],
          public* |> __30 : [t |> t_27 : Pi __28:<>. Type]]
val it = () : unit
```

6.5 Open modules:

```
- Top.elab_string "structure S = struct val i = 5 end \
=                \ open S                             \
=                \ val j = i";
  |- structure S = struct val i = 5 end
     open S
     val j = i
 ~~> S |> S_34 = [i |> i_33 = 5], 1* |> S_35 = S_34, j |> j_37 = S_35.i
   : S |> S_34 : [i |> i_33 : int],
     1* |> S_35 : [i |> i_33 : int],
     j |> j_37 : int
```

6.6 Identifier shadowing:

```
- Top.elab_string "structure Shadow = \
=                 \  struct           \
=                 \    val x = 5      \
```

```
=                 \    val x = true   \
=                 \    val x = nil    \
=                 \  end";
  |- structure Shadow = struct val x = 5 val x = true val x = nil end
 ~~> Shadow |> Shadow_44 =
         [x:1 |> x_39 = 5,
          x:0 |> x_41 = true,
          x |> x_43 = /\ a_42:<Type>. nil [#1 a_42]]
   : Shadow |> Shadow_44 :
         [x:1 |> x_39 : int,
          x:0 |> x_41 : bool,
          x |> x_43 : All a_42:<Type>. list (#1 a_42)]
val it = () : unit

(* no shadowing for different syntactic classes *)
- Top.elab_string "structure NoShadow =          \
=                 \   struct                     \
=                 \     type t = int             \
=                 \     val t = 5                \
=                 \     structure t = struct end  \
=                 \   end                        \
=                 \ type u = NoShadow.t          \
=                 \ val u = NoShadow.t           \
=                 \ structure u = NoShadow.t";
  |- structure NoShadow =
         struct type t = int val t = 5 structure t = struct end end
     type u = NoShadow.t
     val u = NoShadow.t
     structure u = NoShadow.t
 ~~> NoShadow |> NoShadow_21 =
         [t |> t_15 = \ var_16:<>. int, t |> t_19 = 5, t |> t_20 = []],
     u |> u_24 = \ var_25:<>. NoShadow_21.t <>,
     u |> u_28 = NoShadow_21.t,
     u |> u_29 = NoShadow_21.t
   : NoShadow |> NoShadow_21 :
         [t |> t_15 : Pi var_17:<>. S(int), t |> t_19 : int, t |> t_20 : []],
     u |> u_24 : Pi var_26:<>. S(NoShadow_21.t <>),
     u |> u_28 : int,
     u |> u_29 : []
val it = () : unit

(* identifier lookup finds different paths for different syntactic classes *)
- Top.elab_string "type t = int  val t = 5  structure t = struct end \
=                 \ type u = t    val u = t  structure u = t";
  |- type t = int
     val t = 5
     structure t = struct end
     type u = t
     val u = t
     structure u = t
 ~~> t |> t_93 = \ var_94:<>. int,
     t |> t_97 = 5,
```

```
     t |> t_98 = [],
     u |> u_101 = \ var_102:<>. t_93 <>,
     u |> u_105 = t_97,
     u |> u_106 = t_98
   : t |> t_93 : Pi var_95:<>. S(int),
     t |> t_97 : int,
     t |> t_98 : [],
     u |> u_101 : Pi var_103:<>. S(t_93 <>),
     u |> u_105 : int,
     u |> u_106 : []
val it = () : unit
```

6.7 Coercive signature matching:

```
- Top.elab_string
=   "structure Coerce = struct val x = 5 val y = x + 7 val z = y * 12 end \
=  \ structure Coerced : sig val z : int val y : int end = Coerce";
  |- structure Coerce = struct val x = 5 val y = x + 7 val z = y * 12 end
     structure Coerced = Coerce : sig val z : int val y : int end
 ~~> Coerce |> Coerce_73 =
         [x |> x_68 = 5, y |> y_70 = x_68 + 7, z |> z_72 = y_70 * 12],
     Coerced |> Coerced_76 =
         [z |> z_74 = Coerce_73.z, y |> y_75 = Coerce_73.y]
   : Coerce |> Coerce_73 :
         [x |> x_68 : int, y |> y_70 : int, z |> z_72 : int],
     Coerced |> Coerced_76 : [z |> z_74 : int, y |> y_75 : int]
val it = () : unit

- Top.elab_string
=   "structure Coerce : sig val z : int  val y : int end = \
=   \  struct            \
=   \    val x = 5       \
=   \    val y = x + 7     \
=   \    val z = y * 12    \
=   \  end";
  |- structure Coerce =
         struct val x = 5 val y = x + 7 val z = y * 12 end
           : sig val z : int val y : int end
 ~~> Coerce |> Coerce_117 =
         [private |> s_113 =
              [%module_6 |> %module_6_112 =
                   [x |> x_107 = 5,
                    y |> y_109 = x_107 + 7,
                    z |> z_111 = y_109 * 12]],
          public* |> __116 =
              [z |> z_114 = s_113.%module_6.z, y |> y_115 = s_113.%module_6.y]]
   : Coerce |> Coerce_117 :
         [private |> s_113 :
              [%module_6 |> %module_6_112 :
                   [x |> x_107 : int, y |> y_109 : int, z |> z_111 : int]],
          public* |> __116 : [z |> z_114 : int, y |> y_115 : int]]
val it = () : unit
```

A IL static semantics

A.1 Kind formation

$\boxed{\Gamma \vdash knd : \mathbf{Kind}}$

$$\frac{}{\Gamma \vdash \Omega : \mathbf{Kind}} \tag{A.1}$$

$$\frac{\Gamma \vdash knd_1 : \mathbf{Kind} \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd_1 \vdash knd_2 : \mathbf{Kind}}{\Gamma \vdash \Pi\alpha{:}knd_1.\, knd_2 : \mathbf{Kind}} \tag{A.2}$$

$$\frac{\Gamma \vdash knd_1 : \mathbf{Kind} \qquad \ldots \qquad \Gamma \vdash knd_n : \mathbf{Kind}}{\Gamma \vdash \langle knd_1 \times \ldots \times knd_n \rangle : \mathbf{Kind}} \tag{A.3}$$

$$\frac{\Gamma \vdash con : \Omega}{\Gamma \vdash \mathcal{S}(con) : \mathbf{Kind}} \tag{A.4}$$

A.2 Kind equivalence

$\boxed{\Gamma \vdash knd_1 \equiv knd_2 : \mathbf{Kind}}$

$$\frac{}{\Gamma \vdash \Omega \equiv \Omega : \mathbf{Kind}} \tag{A.5}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : \Omega}{\Gamma \vdash \mathcal{S}(con_1) \equiv \mathcal{S}(con_2) : \mathbf{Kind}} \tag{A.6}$$

$$\frac{\Gamma \vdash knd_1 \equiv knd_2 : \mathbf{Kind} \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd_1 \vdash knd'_1 \equiv knd'_2 : \mathbf{Kind}}{\Gamma \vdash \Pi\alpha{:}knd_1.\, knd'_1 \equiv \Pi\alpha{:}knd_2.\, knd'_2 : \mathbf{Kind}} \tag{A.7}$$

$$\frac{\Gamma \vdash knd_1 \equiv knd'_1 : \mathbf{Kind} \qquad \ldots \qquad \Gamma \vdash knd_n \equiv knd'_n : \mathbf{Kind}}{\Gamma \vdash \langle knd_1, \ldots, knd_n \rangle \equiv \langle knd'_1, \ldots, knd'_n \rangle : \mathbf{Kind}} \tag{A.8}$$

A.3 Subkinding

$\boxed{\Gamma \vdash knd_1 \leq knd_2}$

$$\frac{}{\Omega \vdash \Omega \leq} \tag{A.9}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : \Omega}{\Gamma \vdash \mathcal{S}(con_1) \leq \mathcal{S}(con_2)} \tag{A.10}$$

$$\frac{\Gamma \vdash con : \Omega}{\Gamma \vdash \mathcal{S}(con) \leq \Omega} \tag{A.11}$$

$$\frac{\Gamma \vdash knd_2 \leq knd_1 \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd_2 \vdash knd'_1 \leq knd'_2 \qquad \Gamma, \alpha{:}knd_1 \vdash knd'_1 : \mathbf{Kind}}{\Gamma \vdash \Pi\alpha{:}knd_1.\, knd'_1 \leq \Pi\alpha{:}knd_2.\, knd'_2} \tag{A.12}$$

$$\frac{\Gamma \vdash knd_1 \leq knd'_1 \qquad \ldots \qquad \Gamma \vdash knd_n \leq knd'_n}{\Gamma \vdash \langle knd_1 \times \ldots \times knd_n \rangle \leq \langle knd'_1 \times \ldots \times knd'_n \rangle} \tag{A.13}$$

A.4 Constructor kinding

$\boxed{\Gamma \vdash con : knd}$

$$\frac{(\Gamma(\alpha) = knd)}{\Gamma \vdash \alpha : knd} \tag{A.14}$$

$$\frac{}{\Gamma \vdash \mathsf{unit} : \Omega} \tag{A.15}$$

$$\frac{}{\Gamma \vdash \mathsf{bool} : \Omega} \tag{A.16}$$

$$\frac{}{\Gamma \vdash \mathsf{int} : \Omega} \tag{A.17}$$

$$\frac{}{\Gamma \vdash \mathsf{char} : \Omega} \tag{A.18}$$

$$\frac{}{\Gamma \vdash \mathsf{string} : \Omega} \tag{A.19}$$

$$\frac{\Gamma \vdash con_1 : \Omega \qquad \Gamma \vdash con_2 : \Omega}{\Gamma \vdash con_1 \times con_2 : \Omega} \tag{A.20}$$

$$\frac{\Gamma \vdash con : \Omega}{\Gamma \vdash \mathsf{list}\ con : \Omega} \tag{A.21}$$

$$\frac{\Gamma \vdash con_1 : \Omega \qquad \Gamma \vdash con_2 : \Omega}{\Gamma \vdash con_1 \to con_2 : \Omega} \tag{A.22}$$

$$\frac{\Gamma \vdash knd : \mathbf{Kind} \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd \vdash con : \Omega}{\Gamma \vdash \forall \alpha{:}knd.\ con : \Omega} \tag{A.23}$$

$$\frac{\Gamma \vdash knd : \mathbf{Kind} \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd \vdash con : knd'}{\Gamma \vdash \lambda \alpha{:}knd.\ con : \Pi \alpha{:}knd.\ knd'} \tag{A.24}$$

$$\frac{\Gamma \vdash con_1 : \Pi \alpha{:}knd.\ knd' \qquad \Gamma \vdash con_2 : knd}{\Gamma \vdash con_1\ con_2 : [con_2/\alpha]knd'} \tag{A.25}$$

$$\frac{\Gamma \vdash con_1 : knd_1 \qquad \ldots \qquad \Gamma \vdash con_n : knd_n}{\Gamma \vdash \langle con_1, \ldots, con_n \rangle : \langle knd_1 \times \ldots \times knd_n \rangle} \tag{A.26}$$

$$\frac{\Gamma \vdash con : \langle knd_1 \times \ldots \times knd_n \rangle \qquad (1 \le i \le n)}{\Gamma \vdash \pi_i\ con : knd_i} \tag{A.27}$$

$$\frac{\Gamma \vdash mod_{\mathrm{v}} : [lab_1 \triangleright var_1{:}class_1, \ldots, lab_n \triangleright var_n{:}class_n, lab \triangleright \alpha{:}knd, sdecs]}{\Gamma \vdash mod_{\mathrm{v}}.lab : [mod_{\mathrm{v}}.lab_1, \ldots mod_{\mathrm{v}}.lab_n/var_1, \ldots, var_n]knd} \tag{A.28}$$

$$\frac{\Gamma \vdash con : \Omega}{\Gamma \vdash con : \mathcal{S}(con)}\ (\text{K-Sing}) \tag{A.29}$$

$$\frac{\Gamma \vdash con : \Pi \alpha{:}knd.\ knd'' \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd \vdash con\ \alpha : knd'}{\Gamma \vdash con : \Pi \alpha{:}knd.\ knd'}\ (\text{K-}\Pi\text{-Ext}) \tag{A.30}$$

$$\frac{\Gamma \vdash \pi_1\ con : knd_1 \qquad \ldots \qquad \Gamma \vdash \pi_n\ con : knd_n}{\Gamma \vdash con : \langle knd_1 \times \cdots \times knd_n \rangle}\ (\text{K-}\times\text{-Ext}) \tag{A.31}$$

$$\frac{\Gamma \vdash con : knd' \qquad \Gamma \vdash knd' \le knd}{\Gamma \vdash con : knd}\ (\text{K-Sub}) \tag{A.32}$$

A.5 Constructor equivalence

$$\boxed{\Gamma \vdash con_1 \equiv con_2 : knd}$$

$$\frac{\Gamma \vdash con : knd}{\Gamma \vdash con \equiv con : knd}\ (\text{Eq-Refl}) \tag{A.33}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : knd}{\Gamma \vdash con_2 \equiv con_1 : knd}\ (\text{Eq-Symm}) \tag{A.34}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : knd \qquad \Gamma \vdash con_2 \equiv con_3 : knd}{\Gamma \vdash con_1 \equiv con_3 : knd}\ (\text{Eq-Trans}) \tag{A.35}$$

$$\frac{\Gamma \vdash con_1 \equiv con_1' : \Omega \qquad \Gamma \vdash con_2 \equiv con_2' : \Omega}{\Gamma \vdash con_1 \times con_2 \equiv con_1' \times con_2' : \Omega} \tag{A.36}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : \Omega}{\Gamma \vdash \mathsf{list}\, con_1 \equiv \mathsf{list}\, con_2 : \Omega} \tag{A.37}$$

$$\frac{\Gamma \vdash con_1 \equiv con_1' : \Omega \qquad \Gamma \vdash con_2 \equiv con_2' : \Omega}{\Gamma \vdash con_1 \to con_2 \equiv con_1' \to con_2' : \Omega} \tag{A.38}$$

$$\frac{(\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma \vdash knd_1 \equiv knd_2 : \mathbf{Kind} \qquad \Gamma, \alpha{:}knd_1 \vdash con_1 \equiv con_2 : \Omega}{\Gamma \vdash \forall \alpha{:}knd_1.\, con_1 \equiv \forall \alpha{:}knd_2.\, con_2 : \Omega} \tag{A.39}$$

$$\frac{(\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma \vdash knd_1 \equiv knd_2 : \mathbf{Kind} \qquad \Gamma, \alpha{:}knd_1 \vdash con_1 \equiv con_2 : knd'}{\Gamma \vdash \lambda \alpha{:}knd_1.\, con_1 \equiv \lambda \alpha{:}knd_2.\, con_2 : \Pi \alpha{:}knd_1.\, knd'} \tag{A.40}$$

$$\frac{\Gamma \vdash con_1 \equiv con_1' : \Pi \alpha{:}knd.\, knd' \qquad \Gamma \vdash con_2 \equiv con_2' : knd}{\Gamma \vdash con_1\ con_2 \equiv con_1'\ con_2' : [con_2/\alpha]knd'} \tag{A.41}$$

$$\frac{\Gamma \vdash con_1 \equiv con_1' : knd_1 \qquad \ldots \qquad \Gamma \vdash con_n \equiv con_n' : knd_n}{\Gamma \vdash \langle con_1, \ldots, con_n \rangle \equiv \langle con_1', \ldots, con_n' \rangle : \langle knd_1 \times \ldots \times knd_n \rangle} \tag{A.42}$$

$$\frac{\Gamma \vdash con \equiv con' : \langle knd_1 \times \ldots \times knd_n \rangle \qquad (1 \le i \le n)}{\Gamma \vdash \pi_i\, con \equiv \pi_i\, con' : knd_i} \tag{A.43}$$

$$\frac{\begin{array}{c}(mod_{\mathrm{v}} = [lab_1 \triangleright var_1{=}class_1, \ldots, lab_n \triangleright var_n{=}class_n, lab \triangleright \alpha{=}con, sdecs]) \\ \Gamma \vdash mod_{\mathrm{v}}.lab : knd\end{array}}{\Gamma \vdash mod_{\mathrm{v}}.lab \equiv [mod_{\mathrm{v}}.lab_1, \ldots, mod_{\mathrm{v}}.lab_n/var_1, \ldots, var_n]con : knd}\ (\text{Eq-}[]\text{-}\beta) \tag{A.44}$$

$$\frac{(\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd \vdash con' : knd' \qquad \Gamma \vdash con : knd}{\Gamma \vdash (\lambda \alpha{:}knd.\, con')\ con \equiv [con/\alpha]con' : [con/\alpha]knd'}\ (\text{Eq-}\Pi\text{-}\beta) \tag{A.45}$$

$$\frac{\Gamma \vdash con_i : knd_i}{\Gamma \vdash \pi_i \langle con_1, \ldots, con_n \rangle \equiv con_i : knd_i}\ (\text{Eq-}\times\text{-}\beta) \tag{A.46}$$

$$\frac{\Gamma \vdash con : \mathcal{S}(con')}{\Gamma \vdash con \equiv con' : \mathcal{S}(con')}\ (\text{Eq-Sing}) \tag{A.47}$$

$$\frac{\begin{array}{c}\Gamma \vdash con_1 : \Pi \alpha{:}knd.\, knd_1 \qquad \Gamma \vdash con_2 : \Pi \alpha{:}knd.\, knd_2 \\ (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd \vdash con_1\ \alpha \equiv con_2\ \alpha : knd'\end{array}}{\Gamma \vdash con_1 \equiv con_2 : \Pi \alpha{:}knd.\, knd'}\ (\text{Eq-}\Pi\text{-Ext}) \tag{A.48}$$

$$\frac{\Gamma \vdash \pi_1\ con \equiv \pi_1\ con' : knd_1 \qquad \ldots \qquad \Gamma \vdash \pi_n\ con \equiv \pi_n\ con' : knd_n}{\Gamma \vdash con \equiv con' : \langle knd_1 \times \cdots \times knd_n \rangle}\ (\text{Eq-}\times\text{-Ext}) \tag{A.49}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : knd' \qquad \Gamma \vdash knd' \leq knd}{\Gamma \vdash con_1 \equiv con_2 : knd}\ (\text{Eq-Sub}) \tag{A.50}$$

A.6 Expression typing

$\boxed{\Gamma \vdash exp : con}$

$$\frac{(\Gamma(x) = con)}{\Gamma \vdash x : con} \tag{A.51}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \tag{A.52}$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \tag{A.53}$$

$$\frac{\Gamma \vdash exp_1 : \mathsf{bool} \qquad \Gamma \vdash exp_2 : con \qquad \Gamma \vdash exp_3 : con}{\Gamma \vdash \textbf{if}\ exp_1\ \textbf{then}\ exp_2\ \textbf{else}\ exp_3 : con} \tag{A.54}$$

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \tag{A.55}$$

$$\frac{}{\Gamma \vdash \bar{n} : \mathsf{int}} \tag{A.56}$$

$$\frac{}{\Gamma \vdash \text{“}string\text{”} : \mathsf{string}} \tag{A.57}$$

$$\frac{}{\Gamma \vdash \text{‘}c\text{’} : \mathsf{char}} \tag{A.58}$$

$$\frac{O_1(unop) = (con_1, con_2, con_r) \qquad \Gamma \vdash exp_1 : con_1 \qquad \Gamma \vdash exp_2 : con_2}{\Gamma \vdash exp_1\ unop\ exp_2 : con_r} \tag{A.59}$$

$$\frac{O_2(binop) = (con_1, con_2, con_r) \qquad \Gamma \vdash exp_1 : con_1 \qquad \Gamma \vdash exp_2 : con_2}{\Gamma \vdash exp_1\ binop\ exp_2 : con_r} \tag{A.60}$$

$$\frac{\Gamma \vdash exp_1 : con_1 \qquad \Gamma \vdash exp_2 : con_2}{\Gamma \vdash (exp_1, exp_2) : con_1 \times con_2} \tag{A.61}$$

$$\frac{\Gamma \vdash exp : con_1 \times con_2}{\Gamma \vdash \pi_1\ exp : con_1} \tag{A.62}$$

$$\frac{\Gamma \vdash exp : con_1 \times con_2}{\Gamma \vdash \pi_2\ exp : con_2} \tag{A.63}$$

$$\frac{\Gamma \vdash con : \Omega}{\Gamma \vdash \mathsf{nil}\,[con] : \mathsf{list}\ con} \tag{A.64}$$

$$\frac{\Gamma \vdash exp_1 : con \qquad \Gamma \vdash exp_2 : \mathsf{list}\ con}{\Gamma \vdash exp_1 :: exp_2 : \mathsf{list}\ con} \tag{A.65}$$

$$\frac{\Gamma \vdash exp : \mathsf{list}\ con \qquad \Gamma \vdash exp_n : con' \qquad (x, xs \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, x{:}con, xs{:}\mathsf{list}\ con \vdash exp_c : con'}{\Gamma \vdash \mathbf{case}\ exp\ \mathbf{of}\ \mathsf{nil} \Rightarrow exp_n \mid x :: xs \Rightarrow exp_c :} \tag{A.66}$$

$$\frac{(f, x \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, f{:}con_1 \to con_2, x{:}con_1 \vdash exp : con_2}{\Gamma \vdash \mathbf{fix}\ f\ (x{:}con_1) : con_2 \Rightarrow exp : con_1 \to con_2} \tag{A.67}$$

$$\frac{\Gamma \vdash exp_1 : con \to con' \qquad \Gamma \vdash exp_2 : con}{\Gamma \vdash exp_1\ exp_2 : con'} \tag{A.68}$$

$$\frac{\Gamma \vdash knd : \mathbf{Kind} \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd \vdash exp : con}{\Gamma \vdash \Lambda\alpha{:}knd.\ exp : \forall\alpha{:}knd.\ con} \tag{A.69}$$

$$\frac{\Gamma \vdash exp : \forall\alpha{:}knd.\ con' \qquad \Gamma \vdash con : knd}{\Gamma \vdash exp\,[con] : [con/\alpha]con'} \tag{A.70}$$

$$\frac{\Gamma \vdash mod : sig \qquad (s \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, s{:}sig \vdash exp : con \qquad \Gamma \vdash con : \Omega}{\Gamma \vdash \mathbf{let}\ s = mod\ \mathbf{in}\ exp\ \mathbf{end} : con} \tag{A.71}$$

$$\frac{\Gamma \vdash mod_{\mathrm{v}} : [lab_1 \triangleright var_1{:}class_1, \ldots, lab_n \triangleright var_n{:}class_n, lab \triangleright x{:}con, sdecs]}{\Gamma \vdash mod_{\mathrm{v}}.lab : [mod_{\mathrm{v}}.lab_1, \ldots mod_{\mathrm{v}}.lab_n/var_1, \ldots, var_n]con} \tag{A.72}$$

$$\frac{\Gamma \vdash exp : con' \qquad \Gamma \vdash con' \equiv con : \Omega}{\Gamma \vdash exp : con}\ (\textsc{T-Conv}) \tag{A.73}$$

A.7 Signature formation

$\boxed{\Gamma \vdash sig : \mathbf{Sig}}$

$$\frac{\Gamma \vdash sdecs\ \mathbf{ok}}{\Gamma \vdash [sdecs] : \mathbf{Sig}} \tag{A.74}$$

$\boxed{\Gamma \vdash sdecs\ \mathbf{ok}}$

$$\frac{}{\Gamma \vdash \cdot\ \mathbf{ok}} \tag{A.75}$$

$$\frac{\Gamma, dec \vdash sdecs\ \mathbf{ok}}{\Gamma \vdash lab \triangleright dec, sdecs\ \mathbf{ok}} \tag{A.76}$$

A.8 Module signature matching

$\boxed{\Gamma \vdash mod : sig}$

$$\frac{(\Gamma(s) = sig)}{\Gamma \vdash s : sig} \tag{A.77}$$

$$\frac{\Gamma \vdash sbnds : sdecs}{\Gamma \vdash [sbnds] : [sdecs]} \tag{A.78}$$

$$\frac{\Gamma \vdash mod_{\mathrm{v}} : [lab_1 \triangleright var_1{:}class_1, \ldots, lab_n \triangleright var_n{:}class_n, lab \triangleright s{:}sig, sdecs]}{\Gamma \vdash mod_{\mathrm{v}}.lab : [mod_{\mathrm{v}}.lab_1, \ldots mod_{\mathrm{v}}.lab_n/var_1, \ldots, var_n]sig} \tag{A.79}$$

$$\frac{\Gamma \vdash mod : sig}{\Gamma \vdash (mod :> sig) : sig} \tag{A.80}$$

$$\frac{\Gamma \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright \alpha{:}knd, sdecs'] \qquad \Gamma \vdash mod_{\mathrm{v}}.lab : knd'}{\Gamma \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright \alpha{:}knd', sdecs']}\ (\textsc{Sg-Self}) \tag{A.81}$$

Rule (A.81): Selfification allows us to give module values more precise signatures. In particular, $mod_{\mathrm{v}}.lab$ can be given a singleton kind knd' which will be slotted into mod_{v}'s signature.

$$\frac{\Gamma \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright s{:}sig, sdecs'] \qquad \Gamma \vdash mod_{\mathrm{v}}.lab : sig'}{\Gamma \vdash mod_{\mathrm{v}} : [sdecs, lab \triangleright s{:}sig', sdecs']}\ (\textsc{Sg-Self-Rec}) \tag{A.82}$$

Rule (A.82): Allow recursive application of selfification through sub-modules.

$$\frac{\Gamma \vdash mod : sig' \qquad \Gamma \vdash sig' \leq sig}{\Gamma \vdash mod : sig}\ (\textsc{Sg-Sub}) \tag{A.83}$$

$\boxed{\Gamma \vdash sbnds : sdecs}$

$$\frac{}{\Gamma \vdash \cdot : \cdot} \tag{A.84}$$

$$\frac{\Gamma \vdash bnd : dec \qquad \Gamma, dec \vdash sbnds : sdecs}{\Gamma \vdash lab \triangleright bnd, sbnds : lab \triangleright dec, sdecs} \tag{A.85}$$

$\boxed{\Gamma \vdash bnd : dec}$

$$\frac{\Gamma \vdash exp : con}{\Gamma \vdash x = exp : x{:}con} \tag{A.86}$$

$$\frac{\Gamma \vdash con : knd}{\Gamma \vdash \alpha = con : \alpha{:}knd} \tag{A.87}$$

$$\frac{\Gamma \vdash mod : sig}{\Gamma \vdash s = mod : s{:}sig} \tag{A.88}$$

A.9 Subsignature matching

$\boxed{\Gamma \vdash sig_1 \leq sig_2}$

$$\frac{\Gamma \vdash sdecs_1 \leq sdecs_2}{\Gamma \vdash [sdecs_1] \leq [sdecs_2]} \tag{A.89}$$

$\boxed{\Gamma \vdash sdecs_1 \leq sdecs_2}$

$$\frac{}{\Gamma \vdash \cdot \leq \cdot} \tag{A.90}$$

$$\frac{\Gamma \vdash con_1 \equiv con_2 : \Omega \qquad (x \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, x{:}con_1 \vdash sdecs_1 \leq sdecs_2 \qquad \Gamma, x{:}con_2 \vdash sdecs_2\ \mathbf{ok}}{\Gamma \vdash lab \triangleright x{:}con_1, sdecs_1 \leq lab \triangleright x{:}con_2, sdecs_2} \tag{A.91}$$

$$\frac{\Gamma \vdash knd_1 \leq knd_2 \qquad (\alpha \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, \alpha{:}knd_1 \vdash sdecs_1 \leq sdecs_2 \qquad \Gamma, \alpha{:}knd_2 \vdash sdecs_2\ \mathbf{ok}}{\Gamma \vdash lab \triangleright \alpha{:}knd_1, sdecs_1 \leq lab \triangleright \alpha{:}knd_2, sdecs_2} \tag{A.92}$$

$$\frac{\Gamma \vdash sig_1 \leq sig_2 \qquad (s \notin \mathrm{dom}(\Gamma)) \qquad \Gamma, s{:}sig_1 \vdash sdecs_1 \leq sdecs_2 \qquad \Gamma, s{:}sig_2 \vdash sdecs_2 \ \mathbf{ok}}{\Gamma \vdash lab \triangleright s{:}sig_1, sdecs_1 \leq lab \triangleright s{:}sig_2, sdecs_2} \tag{A.93}$$